

Análisis de Herramientas para Detectar Scripts Maliciosos en Node.js

, Corporación Universitaria Remington

Facultad de Ingeniería en Sistemas

Seminario Ciberseguridad y Defensa

Daniel Andres Restrepo Prado

Ing. Esp. ALFONSO VALENCIA MSc.

2026

Tabla de Contenidos

Resumen.....	4
Objetivo General.....	5
Objetivos Específicos.....	5
Marco Conceptual.....	5
1. Cadena de Suministro de Software	5
2. Node.js y NPM.....	6
3. Scripts de NPM.....	6
4. Paquetes Maliciosos y Código Inseguro	7
4.1 Paquete Malicioso	7
4.2 Script Malicioso	7
4.3 Código Vulnerable	7
4.4 Código Potencialmente Peligroso	7
5. Vulnerabilidades en la Cadena de Suministros NPM	7
5.1 Desarrollador.....	9
5.2 NPM/Repositorio	9
5.3 Paquetes y Dependencias.....	10
5.4 Proyecto	11
5.5 Producción	11
6. Análisis de Seguridad del Código.....	12
6.1 SAST (Static Application Security Testing).....	12
6.2 Análisis de Dependencias	12
6.3 Análisis de Composición de Software (SCA).....	13

6.4 Detección de Patrones Inseguros	13
7. Semgrep	13
8. SonarQube.....	14
9. OWASP Dependency-Check	14
10. OSV-Scanner	14
Marco Contextual.....	15
Desarrollo Experimental y Análisis de Herramientas.....	16
1. Configuración del Entorno de Pruebas y Escenarios	16
2. Ejecución de Escaneos y Recolección de Evidencia	16
3. Análisis Comparativo de Resultados	17
3.1 Tiempos de Ejecución.....	17
3.2 Detección de Vulnerabilidades	19
3.3 Calidad del Reporte.....	21
3.4 Prueba Comparativa de Detección de Vulnerabilidades.....	23
3.5 Facilidad de Ejecución.....	28
Conclusiones	30
Referencias.....	32

Resumen

El objetivo principal de este informe técnico es analizar y compara mediante parámetros definidos de cuatro herramientas de software libre frente a scripts maliciosos y comportamientos anómalos en proyectos de Node.js: Semgrep, SonarQube (análisis estático de código), y OWASP Dependency-Check y OSV-Scanner (análisis de composición de software). A raíz de los recientes incidentes de seguridad que han afectado al ecosistema, en los que la inclusión de código no autorizado ha comprometido la seguridad de múltiples proyectos a nivel global, resulta fundamental determinar qué tipo de amenaza detecta cada enfoque antes del despliegue en producción. Para ello, se ejecutaron las herramientas sobre códigos maliciosos, entre ellos el publicado por el grupo teamPCP, y se compararon su tiempo de respuesta, su capacidad de detección, la calidad de sus reportes y su usabilidad. Los resultados muestran que la detección de scripts maliciosos depende de reglas de análisis estático configuradas por el usuario, mientras que las herramientas de composición de software identifican vulnerabilidades conocidas en las dependencias. Por ello, se recomienda combinar ambos enfoques para reducir el riesgo de ejecutar código malicioso en las etapas de desarrollo.

Palabras clave: Node.js, NPM, cadena de suministro, scripts maliciosos, SAST, SCA

Objetivo General

Analizar y comparar herramientas de software libre y de código abierto que permitan detectar scripts maliciosos y comportamientos anómalos dentro del ecosistema de dependencias de Node.js.

Objetivos Específicos

- Seleccionar cuatro herramientas de código abierto o gratuitas que representen dos enfoques de análisis de seguridad para Node.js: el análisis estático de código (Semgrep y SonarQube) y el análisis de composición de software (OWASP Dependency-Check y OSV-Scanner).
- Comparar el desempeño técnico de las herramientas evaluando el tiempo de respuesta, la capacidad de detección, la claridad del reporte y la facilidad de ejecución, esta última medida según el número de pasos y configuraciones que exige cada herramienta (Tabla 6).
- Determinar qué tipo de amenaza (scripts de instalación maliciosos o vulnerabilidades conocidas) detecta cada enfoque y proponer una combinación de herramientas para reducir el riesgo.

Marco Conceptual

1. Cadena de Suministro de Software

La cadena de suministro de software hace referencia a todos los elementos que intervienen o hacen parte del desarrollo de software: “La cadena de suministro de software consiste en código, configuraciones, binarios propietarios y de código abierto, librerías, plugins y dependencias de contenedores” (Schmitt, 2026).

Hoy en día, gran parte del software depende de componentes externos para funcionar correctamente, lo que en sí mismo puede implicar un riesgo para el desarrollo: “Dada la complejidad y el amplio alcance de las cadenas de suministro de software, existen numerosas formas de introducir cambios no autorizados en el software que entregas a tus usuarios” (Google, 2026).

2. Node.js y NPM

Node.js permite ejecutar código JavaScript sin necesidad de un navegador web. Al respecto, se define como “un entorno de ejecución de Javascript de código abierto y multiplataforma que ejecuta el motor JavaScript V8, el núcleo de Google Chrome, fuera del navegador” (Node.js, s.f.).

“NPM es el gestor de paquetes estándar para Node.js” (Node.js, s.f.). La importancia de NPM en Node.js radica en que gestiona las dependencias de las aplicaciones que se van a desarrollar, lo que permite que funcionen correctamente y ahorra tiempo a los desarrolladores.

3. Scripts de NPM

Los scripts de NPM son fragmentos de código que se ejecutan durante determinadas etapas del ciclo de instalación o desarrollo de un proyecto y que, por lo general, están diseñados para facilitar el trabajo de los programadores. Sin embargo, en ocasiones los repositorios pueden contener scripts maliciosos que buscan captar la información de quien los utiliza.

La propiedad de tu archivo soporta varios scripts integrados y sus eventos predefinidos del ciclo de vida, así como scripts arbitrarios. Todo esto puede ejecutarse corriendo o simplemente ejecutándolo. También se ejecutarán *comandos pre* y *post* con nombres coincidentes para esos. (Karrys, 2022)

4. Paquetes Maliciosos y Código Inseguro

4.1 Paquete Malicioso

Un paquete malicioso es aquel que, al descargarse, integra código capaz de extraer o comprometer la información de las personas o empresas involucradas. Estos paquetes suelen ser difíciles de identificar, incluso para los escáneres de seguridad tradicionales, que tienden a pasarlos por alto, como señala Hartland (2026).

4.2 Script Malicioso

Un script malicioso es aquel que ejecuta comandos de código capaces de, entre otras cosas, interceptar información del usuario, como su localización y su historial de navegación. Estos scripts suelen estar incluidos en repositorios infectados.

4.3 Código Vulnerable

El código vulnerable es aquel que contiene fallos o errores de programación que permiten a los atacantes encontrar puertas de entrada al programa y, de esta forma, extraer información importante.

4.4 Código Potencialmente Peligroso

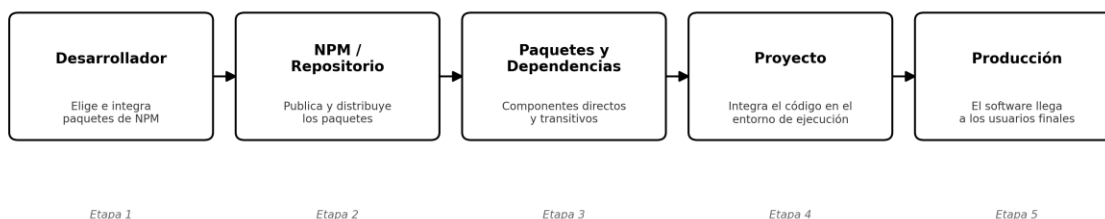
Un código potencialmente peligroso es aquel que contiene instrucciones que pueden llegar a comprometer tanto la seguridad como la privacidad de los usuarios finales de un proyecto de software.

5. Vulnerabilidades en la Cadena de Suministros NPM

Para ejemplificar la cadena de suministro de software dentro del ecosistema NPM, esta puede representarse como se muestra en la Figura 1

Figura 1

Cadena de suministro de software en el ecosistema NPM



Nota: En esta representación se identifican las etapas por las que pasan los componentes de terceros antes de formar parte de un proyecto de software. Cada etapa corresponde a un elemento de la cadena de suministro que puede representar riesgos de seguridad distintos y que, si se identifican a tiempo, evitan que el problema llegue al entorno de producción.

Fuente: Elaboración propia

5.1 Desarrollador

El desarrollador es la persona, o el equipo de desarrollo, encargada de programar el proyecto de software. Es quien decide qué herramientas se utilizarán durante el desarrollo, como frameworks y bibliotecas, y dentro de estas decisiones se encuentra la selección de paquetes de NPM para incorporar funcionalidades específicas al proyecto. “Un desarrollador también gestiona otras tareas relacionadas con la creación, modificación y gestión de software, como la documentación del software, la arquitectura, las bases de datos y la experiencia del usuario” (Equipo editorial de Indeed, 2024).

La elección de estos componentes es muy importante para la seguridad de la cadena de suministro, dado que el desarrollador deposita cierto nivel de confianza en código que no necesariamente fue desarrollado por su equipo. Por lo tanto, es necesario tener en cuenta su procedencia, su mantenimiento y su seguridad.

5.2 NPM/Repositorio

NPM es uno de los principales ecosistemas utilizados para la distribución de paquetes de JavaScript y Node.js. En este ecosistema, los desarrolladores pueden publicar y distribuir paquetes para que otros los incorporen en sus proyectos. Así, el repositorio actúa como intermediario entre quienes publican los paquetes y quienes los utilizan.

Esta es la etapa en la que los desarrolladores enfrentan más amenazas, ya que, al descargar paquetes desde el repositorio, depositan su confianza en los usuarios que los publican. Entre las amenazas más comunes en esta etapa se encuentran el typosquatting (paquetes con nombres muy similares a los de librerías populares, para inducir a un error de escritura), la confusión de dependencias (publicar un paquete público con el mismo nombre que uno interno de una empresa para que sea instalado por error), el secuestro de cuentas de mantenedores legítimos y los scripts de instalación maliciosos incluidos en versiones aparentemente confiables (Hartland, 2026).

5.3 Paquetes y Dependencias

Qué es un Paquete. Los paquetes constituyen una parte fundamental del desarrollo de aplicaciones con Node.js, ya que permiten incorporar funcionalidades desarrolladas previamente y reutilizables en diferentes proyectos. “Los paquetes son los pilares de Node.js proyectos. Son módulos de código reutilizables que puedes integrar en tus aplicaciones para añadir características o funcionalidades específicas” (Askarov, 2023).

Qué es una Dependencia. Una dependencia es la relación mediante la cual interactúan las diferentes bibliotecas dentro de un proyecto de software. Incluir dependencias es una de las prácticas más comunes, ya que facilita y agiliza el desarrollo. “Las dependencias entre librerías

se refieren a la relación que existe entre diferentes bibliotecas o paquetes utilizados en un programa” (Llamas, s.f.).

Las dependencias pueden clasificarse en directas y transitivas. Las directas son las que el desarrollador añade explícitamente a su proyecto: “Las dependencias directas son componentes de software que los desarrolladores eligen explícitamente y declaran para su uso en un proyecto” (Sonatype, s.f.). Estas se declaran, por ejemplo, en archivos como el package.json o el pom.xml. Por otro lado, las dependencias transitivas corresponden a los paquetes de los que dependen otras dependencias para su correcto funcionamiento: “Las dependencias transitivas son componentes de software de los que una aplicación depende indirectamente. No son añadidos directamente por el desarrollador” (Sonatype, s.f.).

Debido a las características de las dependencias, existe el riesgo de que los desarrolladores no conozcan por completo todos los componentes que terminan incorporando a su proyecto, especialmente en el caso de las dependencias transitivas, lo que podría afectar a los proyectos en los que se utilicen.

5.4 Proyecto

En esta etapa, el proyecto deja de estar compuesto únicamente por el código desarrollado internamente e incluye dependencias externas que se integran en el entorno de ejecución y compilación.

Como consecuencia, una vulnerabilidad presente en alguna de estas dependencias podría comprometer la seguridad del proyecto. Por esta razón, es importante controlar las dependencias que se utilizan, así como sus versiones y actualizaciones.

5.5 Producción

La producción es la etapa en la que la aplicación desarrollada se despliega y comienza a ser utilizada por usuarios y otros sistemas. Si en las etapas anteriores no se detectaron las vulnerabilidades, estas podrían llegar hasta aquí y afectar a los usuarios finales. “El entorno en vivo donde los usuarios reales interactúan con la aplicación. Los cambios en producción tienen impacto inmediato, por lo que deben probarse exhaustivamente antes del despliegue. Se abrevia como ‘prod’” (Lafuente, s.f.).

En este sentido, se puede entender que, dentro de la cadena, una aplicación no depende únicamente del código desarrollado por el equipo responsable del proyecto, sino también de factores externos utilizados en su construcción.

6. Análisis de Seguridad del Código

6.1 SAST (Static Application Security Testing)

El SAST consiste en analizar el código fuente, el bytecode o los binarios de una aplicación en busca de vulnerabilidades. Al ser un análisis estático, no requiere ejecutar el código, por lo que permite detectar problemas antes de que comprometan el entorno de desarrollo. “En pocas palabras, es un enfoque de ‘caja blanca’ para identificar fallos de seguridad en las aplicaciones, examinando el código de la aplicación de forma estática (sin ejecutarlo) para detectar posibles amenazas de seguridad” (Sonatype, s.f.).

El SAST identifica patrones y malas prácticas de programación y, con ello, detecta errores y posibles vulnerabilidades en el código. Además, ayuda a mantener un estándar de seguridad que protege todo el ecosistema del software en desarrollo.

Herramientas de análisis estático como SonarQube clasifican sus hallazgos en cuatro tipos: un Bug (error de fiabilidad, código que probablemente falle en tiempo de compilación o de

ejecución), una Code Smell (problema de mantenibilidad que no rompe el programa, pero dificulta su lectura o modificación), una Vulnerability (código que puede ser explotado por un atacante) y un Security Hotspot (código sensible en materia de seguridad que un desarrollador debe revisar manualmente para confirmar si es o no explotable) (SonarSource, s.f.-b).

6.2 Análisis de Dependencias

El análisis de dependencias consiste en identificar los componentes de terceros y de código abierto que utiliza un proyecto y evaluar los riesgos que estos introducen. Según OWASP (s.f.-b), todo componente, ya sea directo o transitivo, es candidato de análisis, pues el riesgo heredado del software de terceros puede afectar directamente a las aplicaciones que dependen de él.

Este análisis es relevante porque los componentes vulnerables o desactualizados figuran entre las diez principales categorías de riesgo de seguridad en aplicaciones web, categoría que abarca tanto los componentes utilizados directamente como las dependencias anidadas (OWASP, 2021).

6.3 Análisis de Composición de Software (SCA)

El análisis de composición de software evalúa los componentes de código abierto de un proyecto y tiene, entre sus principales funciones, detectar vulnerabilidades, controlar las licencias de código abierto y gestionar las dependencias.

El SCA es una metodología de pruebas de seguridad de *software* que identifica, inventaría y evalúa automáticamente los componentes de código abierto y de terceros dentro de una base de código. Estos componentes pueden ser desde librerías y módulos hasta marcos de trabajo e imágenes de contenedores. (Fluid Attacks, 2025)

6.4 Detección de Patrones Inseguros

La detección de patrones inseguros consiste en identificar, dentro del código, construcciones que representan un riesgo para la seguridad, como instrucciones que ejecutan comandos del sistema o que envían información a servidores externos. Herramientas como Semgrep evalúan el código fuente mediante reglas que describen cómo luce el código inseguro y señalan las líneas que coinciden con ellas. Según Semgrep (s.f.), mientras más temprano se realice este análisis en el proceso de desarrollo, más fácil resulta para los equipos detectar estos antipatrones de seguridad.

7. Semgrep

Según Semgrep (s.f.), Semgrep es una herramienta de análisis estático (SAST), rápida y de código abierto, que examina el código, encuentra errores, aplica reglas de seguridad y estándares de codificación, y es compatible con más de 30 lenguajes de programación. En este proyecto se utilizará para analizar, sin ejecutarlo, código con posibles vulnerabilidades y, a partir de los resultados, determinar su efectividad comparándolos con los análisis realizados sobre el mismo código de prueba con otras herramientas.

8. SonarQube

SonarQube es una plataforma que analiza el código de los proyectos durante su desarrollo e informa en tiempo real sobre posibles problemas y vulnerabilidades.

SonarQube funciona integrándose directamente en su entorno de desarrollo y en los procesos de CI/CD para realizar análisis estáticos de su código. A medida que escribe código en su IDE, SonarQube para IDE (el complemento del IDE) realiza análisis en tiempo real para resaltar los problemas de inmediato, ofreciendo explicaciones y

sugerencias de soluciones rápidas adaptadas a su contexto específico. (SonarSource, s.f.-a)

9. OWASP Dependency-Check

"Dependency-Check es un conjunto de herramientas de Análisis de Composición de Software (SCA) que identifica las dependencias de un proyecto y verifica si existen vulnerabilidades conocidas y públicamente divulgadas" (OWASP, s.f.-a). En otras palabras, permite generar análisis de seguridad para prevenir la inclusión de dependencias maliciosas o con vulnerabilidades identificadas que, por esa razón, hayan sido descontinuadas.

10. OSV-Scanner

“Proporciona un formato de datos legible tanto por personas como por máquinas, que describe las vulnerabilidades de manera que correspondan con precisión a las versiones de paquetes de código abierto o a los hashes de commit” (OSV, s.f). Al igual que OWASP Dependency-Check, OSV-Scanner permite detectar las dependencias en las que se han identificado amenazas y, a partir de ello, comparar su facilidad de uso y su efectividad al detectarlas.

Marco Contextual

El desarrollo de aplicaciones modernas viene acompañado del uso de herramientas y componentes desarrollados por terceros, los cuales permiten reducir el tiempo de desarrollo y reutilizar funcionalidades previamente implementadas. En Node.js, esta práctica está estrechamente vinculada al uso de NPM (Node Package Manager), mediante el cual los desarrolladores incorporan a sus proyectos los paquetes y dependencias que requieren para agilizar su trabajo.

Debido a la confianza y al uso constante de NPM, se pudieron pasar por alto riesgos de seguridad, pues un proyecto puede incorporar código que no fue desarrollado directamente por su equipo. Estos paquetes podrían contener vulnerabilidades, configuraciones inseguras o comportamientos maliciosos que representan una gran amenaza para el proyecto.

El 1 de mayo de 2026 se dio a conocer que un grupo de hackers conocido como teamPCP liberó en GitHub una guía detallada para desarrollar malware bajo el nombre de Shai-Hulud. Se sabía que el repositorio contenía una cadena de texto que impedía que el servicio de Claude Code analizara los perfiles de los responsables.

Según Bustan (2026), tras analizar el código, se encontró que este subía credenciales robadas a un repositorio de GitHub, enviaba información a un servidor establecido y extraía credenciales, billeteras de criptomonedas y todo tipo de información de cuentas de usuario.

Debido a esto, la confianza en los paquetes de NPM se vio fuertemente afectada: surgieron recomendaciones de dejar de usar los repositorios de NPM y artículos que presentaron los hechos como una crisis de seguridad sin precedentes, como lo expuso Brad LaPorte (2025) en su artículo:

Shai-Hulud debería acaparar los titulares — no como un incidente con proveedores, sino como una crisis total de seguridad en la cadena de suministro. Expone las vulnerabilidades sistémicas en el ecosistema de código abierto, la fragilidad de las credenciales de los desarrolladores y la urgente necesidad de proteger las canalizaciones.

Desarrollo Experimental y Análisis de Herramientas

1. Configuración del Entorno de Pruebas y Escenarios

Se recopilaron códigos maliciosos para ejecutarlos en los programas de prueba, entre ellos archivos package.json con información de paquetes que son vulnerables y el código proporcionado por teamPCP. Estos códigos contienen scripts que, al ejecutarse, recopilan y envían información a un tercero a través de internet.

2. Ejecución de Escaneos y Recolección de Evidencia

En cada una de las herramientas se ejecutaron los diferentes códigos, se midió el tiempo de respuesta y se analizó el informe que entregaban, haciendo énfasis en si era fácilmente comprensible y en qué tan amplia era la diferencia a la hora de detectar estos scripts maliciosos.

En primera instancia se ejecutaron códigos diseñados específicamente para esta prueba; posteriormente, se realizaron pruebas con códigos cuyas vulnerabilidades son de conocimiento público y, para finalizar, se analizó el código de teamPCP.

3. Análisis Comparativo de Resultados

3.1 Tiempos de Ejecución

En la Tabla 1 se observa que, entre las herramientas de análisis estático, Semgrep obtuvo mejores tiempos que SonarQube desde el primer escaneo (6 y 17 segundos, respectivamente). Entre las herramientas de composición de software, OSV-Scanner fue la más rápida, con 3 segundos en cada ejecución, mientras que OWASP Dependency-Check requirió más tiempo en su primer intento (17 segundos), lo cual es esperable, ya que debe consultar y actualizar bases de datos externas. También se observa una disminución significativa en su segundo escaneo (7 segundos), que redujo el tiempo a menos de la mitad, aunque se mantiene por encima del de OSV-Scanner. Cabe aclarar que las herramientas de composición de software analizaron el

paquete lodash 4.17.15, mientras que las de análisis estático analizaron el script de teamPCP, por lo que la comparación de tiempos entre ambos grupos es solo referencial.

Tabla 1

Tiempo de ejecución de cada herramienta al analizar el script de teamPCP

Tipo de herramienta	Tiempo de ejecución 1	Tiempo de ejecución 2	Tiempo de ejecución 3	Tiempo promedio	Tipo de procesamiento
Semgrep	6 segundos	5 segundos	5 segundos	5,33 segundos	Análisis de script teamPCP
OWASP	17 segundos	7 segundos	6 segundos	10 segundos	Análisis de paquete lodash 4.17.15
OSV-Scanner	3 segundos	3 segundos	3 segundos	3 segundos	Análisis de paquete lodash 4.17.15
SonarQube	17 segundos	15 segundos	14 segundos	15,33 segundos	Análisis de script teamPCP

Nota. Esta tabla muestra el tiempo que tardó cada herramienta en entregar su informe.

3.2 Detección de Vulnerabilidades

Como se observa en la Tabla 2, las herramientas presentan diferencias importantes en la detección, según el tipo de análisis que realiza cada una. En el caso de Semgrep, es obligatorio configurar el archivo YAML para que pueda analizar el script postinstall, y las 30 vulnerabilidades detectadas provienen también de código que se encontraba por fuera de los archivos package.json que hacían parte del proyecto de análisis.

Por otra parte, SonarQube detectó 12 vulnerabilidades, es decir, 18 menos que Semgrep. También se observa una diferencia considerable entre OWASP Dependency-Check y OSV-Scanner en el número de amenazas detectadas.

Tabla 2*Comparación de la detección de scripts y vulnerabilidades por herramienta*

Criterio	Semgrep con regla (YAML)	OWASP	OSV-Scanner	SonarQube
Tipo de análisis	Híbrido (SAST + configuración YAML)	Contrasta las versiones de las librerías contra bases de datos de vulnerabilidades conocidas	Contrasta las versiones de las librerías contra bases de datos de vulnerabilidades conocidas	Análisis mediante SonarScanner
Detección de scripts	Alta: al configurar el archivo YAML, detecta el postinstall y comandos sospechosos del código	Detectó el paquete con sus vulnerabilidades, su CVE correspondiente y un enlace a su descripción	Detectó el paquete con sus vulnerabilidades y suministra el código correspondiente a cada una	Media: se observó una diferencia significativa frente a Semgrep en esta prueba
Detección de vulnerabilidades	Limitada: se requiere configurar el archivo YAML. Se detectaron 30 vulnerabilidades	Alta y automatizada: revisa en red y bases de datos actualizadas. Se detectaron 18 vulnerabilidades catalogadas como críticas	Se detectó el paquete malicioso y 4 vulnerabilidades (2 de riesgo alto y 2 de riesgo medio)	Solo con ejecutar el comando del escáner se detectaron 12 vulnerabilidades

Nota. Esta tabla compara la detección de scripts inseguros y de vulnerabilidades conocidas

(CVE) de cada una de las herramientas.

3.3 Calidad del Reporte

Como se muestra en la Tabla 3, en el caso de Semgrep el informe puede resultar complicado de entender por la forma en que se presenta su estructura inicial. Para facilitar su comprensión, se ejecutó un comando en Python que permitió visualizar mejor las vulnerabilidades encontradas junto con su código de identificación, lo que facilita informarse sobre cada una de ellas.

En contraste con Semgrep, OWASP Dependency-Check y OSV-Scanner presentaron informes más detallados y ofrecieron la posibilidad de acceder con un clic a las definiciones de cada vulnerabilidad. SonarQube, por su parte, muestra los resultados en una interfaz web local; sin embargo, su uso resultó poco intuitivo al inicio y no indica con claridad el peligro que representa cada hallazgo.

Tabla 3

Calidad y comprensibilidad de los reportes generados por cada herramienta

Criterio	Semgrep	OWASP Dependency- Check	OSV-Scanner	SonarQube
Tipo de reporte	Se obtuvo un reporte en formato txt donde se exponían las vulnerabilidades encontradas	Se obtuvo un reporte en HTML con todas las vulnerabilidades que se encontraron	Se obtuvo un reporte que era observable desde la terminal de PowerShell	El reporte se puede visualizar desde el localhost y allí se indica la cantidad de amenazas encontradas
Comprensibilidad	En el caso de Semgrep fue necesario ejecutar un comando en Python para poder organizar la información y hacerla más comprensible	El archivo HTML contiene descripciones y especificaciones concretas de cada una de las vulnerabilidades y enlaces que dirigen a las definiciones de estas mismas	Desde la terminal se podía observar cada una de las vulnerabilidades, la versión analizada y la versión en la que se resolvió el problema	SonarQube resultó poco intuitivo en un primer momento y, aunque muestra las amenazas detectadas, no indica con claridad el peligro que representa cada uno de los problemas del código

Nota. Esta tabla compara el tipo de reporte que genera cada herramienta y su comprensibilidad.

3.4 Prueba Comparativa de Detección de Vulnerabilidades

Como se muestra en la Tabla 4, se observan grandes diferencias entre OWASP Dependency-Check y OSV-Scanner en la detección de vulnerabilidades, siendo en ocasiones una más del doble que la otra, y con una calificación distinta de la gravedad de cada vulnerabilidad. Se analizaron los Package.json de cada uno de los proyectos que se mencionan en la tabla.

Tabla 4

Comparación de las vulnerabilidades detectadas por OWASP Dependency-Check y OSV-Scanner por repositorio

Repositorio	Tipo	OWASP Dependency- Check	OSV-Scanner	Análisis de resultados
Juice Shop	Proyecto grande/Complejo	45 vulnerabilidades encontradas, críticas 15, altas 23, 7 moderadas	80 vulnerabilidades detectadas 5 críticas, 41 altas, 24 moderadas, 7 desconocidas y 3 bajas	En este caso, OSV-Scanner detectó más vulnerabilidades; sin embargo, OWASP Dependency-Check presentó más problemas para ejecutar el análisis
Goof	Vulnerabilidades Supply chain	114 vulnerabilidades encontradas, 20 críticas, 56 altas 38 moderadas	285 vulnerabilidades encontradas 43 críticas, 114 altas, 97 moderadas, 12 bajas y 19 desconocidas	En este caso, OSV-Scanner detecta más del doble de vulnerabilidades que OWASP Dependency-Check
DNVA	App node.js insegura	56 vulnerabilidades encontradas 23 críticas, 9 moderadas 24 altas	24 vulnerabilidades 2 bajo, 4 moderados, 8 altas, 10 críticas	Se pudieron notar diferencias considerables; en este caso fue más efectivo OWASP Dependency-Check

Repositorio	Tipo	OWASP Dependency- Check	OSV-Scanner	Análisis de resultados
Local	Control manual	34 vulnerabilidades encontradas 10 críticas, 20 moderadas, 4 altas	22 vulnerabilidades encontradas, 5 críticas, 10 moderadas y 7 altas	Se hizo una prueba de forma local con un proyecto diseñado para ver cuántas vulnerabilidades encontraba cada herramienta

Nota. Esta tabla compara únicamente la detección de vulnerabilidades de estas dos herramientas.

La Tabla 5 compara los hallazgos de Semgrep y SonarQube. A diferencia de OWASP Dependency-Check y OSV-Scanner, estas herramientas analizan el código fuente y no las versiones de las dependencias, por lo que sus hallazgos corresponden a patrones y defectos del código y no a vulnerabilidades conocidas (CVE) de paquetes.

Tabla 5*Comparación de los hallazgos de Semgrep y SonarQube por repositorio*

Repositorio	Tipo	Semgrep	SonarQube	Análisis de resultados
teamPCP	Script malicioso	30 vulnerabilidades (con reglas YAML)	12 vulnerabilidades	Semgrep detectó 18 hallazgos más que SonarQube sobre el mismo código; SonarQube solo requirió ejecutar el escáner, sin configuración adicional
Juice Shop	Proyecto grande/comp lejo	69 hallazgos (de 1,033 archivos analizados)	Problemas de seguridad: 76; de fiabilidad: 137; de facilidad de mantenimiento: 468	Llama la atención que SonarQube, a pesar de reportar estos problemas, dictaminó que el código pasa la prueba de calidad, mientras que Semgrep resalta sus hallazgos como una invitación a revisar el código antes de enviarlo a producción
Goof	Vulnerabilidades supply chain	28 hallazgos (de 54 archivos analizados)	Problemas de seguridad: 6; de fiabilidad: 122; de facilidad de mantenimiento: 275	A pesar de los problemas de seguridad y de código detectados, SonarQube indica que el proyecto pasa la prueba de calidad, mientras que Semgrep destaca sus 69 hallazgos como señal de que el código debe revisarse antes de producción
DNVA	App Node.js insegura	64 hallazgos (de 129 archivos analizados)	Problemas de seguridad: 7; de fiabilidad: 96; de facilidad de mantenimiento: 96	Semgrep destacó 64 hallazgos que podrían representar amenazas graves para la seguridad del proyecto, mientras que SonarQube detectó 7 problemas de seguridad y, aun así, indicó que el proyecto pasa la prueba de calidad

Nota. Esta tabla compara los hallazgos de Semgrep y SonarQube, herramientas que analizan el código fuente y no las versiones de las dependencias.

3.5 Facilidad de Ejecución

La Tabla 6 sintetiza la facilidad de ejecución de cada herramienta, entendida como la cantidad de pasos y configuraciones adicionales que exige antes de obtener un resultado interpretable. Esta medición se construye a partir de lo ya descrito en las Tablas 1 a 3: OWASP Dependency-Check y OSV-Scanner solo requieren instalar la herramienta y ejecutar un comando; Semgrep necesita, además, un archivo de reglas en YAML para detectar el script postinstall; y SonarQube requiere levantar un servidor local, instalar SonarScanner CLI y generar un token antes de poder ejecutar el escaneo.

Tabla 6

Facilidad de ejecución de cada herramienta

Herramienta	Pasos antes del primer análisis	Configuración adicional	Facilidad de ejecución
Semgrep	Instalar la herramienta y ejecutar el comando de escaneo	Requiere un archivo YAML con reglas propias para detectar el script postinstall	Media
OWASP Dependency-Check	Instalar la herramienta y ejecutar el comando de escaneo	Solicita un token; actualiza sus bases de datos automáticamente	Alta
OSV-Scanner	Instalar la herramienta y ejecutar el comando de escaneo	Ninguna	Alta
SonarQube	Se requiere iniciar el sonar.bat desde la línea de comandos e iniciar sesión desde el localhost puerto 9000	Requiere instalar adicionalmente SonarScanner CLI para escanear los archivos y crear un token que permita la ejecución del escaneo	Baja

Nota. Esta tabla resume el número de pasos y configuraciones que exige cada herramienta antes de obtener un resultado interpretable, a partir de lo descrito en las secciones anteriores.

Conclusiones

El análisis comparativo permitió establecer que las cuatro herramientas evaluadas responden a problemas distintos dentro de la seguridad de la cadena de suministro de Node.js. Semgrep y SonarQube, de análisis estático, examinan el código y los scripts del proyecto, mientras que OWASP Dependency-Check y OSV-Scanner, de composición de software, contrastan las versiones de las dependencias con bases de datos de vulnerabilidades conocidas.

En la detección de scripts maliciosos, objetivo central de este trabajo, Semgrep identificó el script postinstall y los comandos sospechosos de teamPCP solo tras configurar reglas en YAML, mientras que SonarQube lo hizo sin configuración adicional, aunque con menos hallazgos (12 frente a 30). OWASP Dependency-Check y OSV-Scanner no analizan el contenido de los scripts; OSV-Scanner solo detectó el paquete por estar ya catalogado como malicioso (OpenSSF, 2026). En los demás repositorios (Tabla 5), esta diferencia se mantuvo: SonarQube reportó más problemas de fiabilidad y mantenibilidad que de seguridad y, aun así, calificó el código como aprobado, mientras que Semgrep concentró sus hallazgos en advertencias de seguridad. Esto confirma que la detección de scripts depende más del tipo de reglas configuradas que de la herramienta en sí.

En cuanto a las vulnerabilidades conocidas de las dependencias (Tabla 4), OWASP Dependency-Check y OSV-Scanner ofrecen una detección automatizada, pero sus resultados difieren de forma notable entre proyectos: OSV-Scanner encontró más en Juice Shop y en Goof, y OWASP Dependency-Check más en DNVA y en el proyecto local. Como ninguna superó a la otra en todos los casos, usar una sola puede dejar amenazas sin identificar.

Respecto al desempeño (Tablas 1, 3 y 6), OSV-Scanner fue la herramienta más rápida y de ejecución más sencilla, seguida de OWASP Dependency-Check, cuyo tiempo se redujo a

menos de la mitad tras el primer escaneo. Semgrep exigió configurar un archivo YAML y procesar su reporte con un comando adicional, y SonarQube fue la más lenta y la que más configuración requirió, al necesitar un servidor propio. OWASP Dependency-Check y OSV-Scanner, además, entregaron los reportes más claros, con enlaces directos a la definición de cada vulnerabilidad.

A partir de estos resultados, se recomienda combinar ambos enfoques: OSV-Scanner y OWASP Dependency-Check para vigilar vulnerabilidades conocidas, y Semgrep configurado con reglas propias para detectar scripts de instalación maliciosos, ya que ninguna herramienta cubre ambos riesgos por sí sola. Como limitación, las pruebas se realizaron sobre un conjunto reducido de proyectos y sin medir falsos positivos ni falsos negativos, por lo que se sugiere que trabajos futuros amplíen la muestra y evalúen estas tasas.

Referencias

- Askarov, O. (14 de octubre de 2023). *All you need to know about NPM and Node packages as a beginner*. Medium. <https://orozaskarov.medium.com/all-you-need-to-know-about-npm-and-packages-as-a-beginner-b6fcea8b3519>
- Bustan, M. S. (12 de marzo de 2026). *Shai-Hulud goes open source: Malware creators leak their own code to GitHub*. OX Security. <https://www.ox.security/blog/shai-hulud-open-source-malware-github/>
- Equipo editorial de Indeed. (2024). *Qué es un desarrollador y en qué se diferencia de un programador*. Indeed. <https://www.indeed.com/orientacion-profesional/como-encontrar-empleo/que-es-desarrollador-diferencia-programador>
- Fluid Attacks. (7 de abril de 2025). *¿Qué es el análisis de composición de software (SCA)?*. <https://fluidattacks.com/es/cybersecurity-essentials/what-is-software-composition-analysis-sca>
- Google. (20 de julio de 2026). *Descripción general de la seguridad de la cadena de suministro de software*. Google Cloud. <https://docs.cloud.google.com/software-supply-chain-security/docs/overview?hl=es-419>
- Hartland, S. (22 de abril de 2026). *Malicious package detection*. Endor Labs. <https://www.endorlabs.com/learn/malicious-package-detection>
- Karrys, L. (25 de octubre de 2022). *Scripts*. npm Docs. <https://docs.npmjs.com/cli/v8/using-npm/scripts>
- Lafuente, R. (s.f.). *Production*. Lingua-e. <https://www.lingua-e.com/es/glosario/production>

LaPorte, B. (2025). *The NPM worm that no one's talking about — but everyone should be.*

Morphisec. <https://www.morphisec.com/blog/crowdstrike-npm-shai-hulud-worm-supply-chain-attack/>

Llamas, L. (s.f.). *¿Qué son las dependencias en programación?*.

<https://www.luisllamas.es/programacion-que-son-las-dependencias/>

Node.js. (s.f.). *Introduction to Node.js.* <http://nodejs.org/learn/getting-started/introduction-to-nodejs>

OpenSSF. (20 de mayo de 2026). *Detecting malicious packages using the OSV API.*

<https://openssf.org/blog/2026/05/20/detecting-malicious-packages-using-the-osv-api/>

OSV. (s.f.). *A distributed vulnerability database for open source.* <https://osv.dev/>

OWASP. (s.f.-a). *OWASP Dependency-Check.* <https://owasp.org/www-project-dependency-check/>

OWASP. (s.f.-b). *V5: Component analysis requirements.* Software Component Verification Standard. <https://scvs.owasp.org/scvs/v5-component-analysis/>

OWASP. (2021). *A06:2021 – Vulnerable and outdated components.* OWASP Top 10.

https://owasp.org/Top10/2021/A06_2021-Vulnerable_and_Outdated_Components/

Schmitt, J. (7 de abril de 2026). *Cadena de suministro de software: qué es y cómo mantenerla segura.* CircleCI. <https://circleci.com/blog/secure-software-supply-chain/>

Semgrep. (s.f.). *semgrep/semgrep* [Repositorio de GitHub].

<https://github.com/semgrep/semgrep#language-support>

SonarSource. (s.f.-a). *SonarQube: combata la imprecisión de la IA y verifique el código de IA.*

<https://www.sonarsource.com/es/products/sonarqube/>

SonarSource. (s.f.-b). *Overview [Reglas de SonarQube]*. Docs SonarSource.

<https://docs.sonarsource.com/sonarqube/latest/user-guide/rules/overview/>

Sonatype. (s.f.). *What are software dependencies?*.

<https://www.sonatype.com/resources/articles/what-are-software-dependencies>